

导语

web应用跟客户端app不一样的地方在于，web应用的所有资源理论上都是边下载边使用，而且浏览器执行javascript代码的性能并不高，所以对于大型web应用（如在线文档编辑器、在线视频/图片编辑器）等性能问题尤为严重，往往涉及到数十甚至上百个优化点。

传说用户等待打开时间超过3秒，就会流失50%的用户。特别对于复杂的编辑器业务，其他业务可能优化后是2秒，不优化是3秒4秒，编辑器业务不优化就是30秒，一分钟都有可能，基本是没法留住用户的。

在进入具体优化措施之前，可以先理解这些原则：

1、不能点对点去做性能优化，想到哪个点去优化哪个点。一是优化项很难找全，二是很难找到当前roi更高的地方。所以先要对整个链路的性能情况进行摸底，并进行抽象分类。

2、性能防腐，性能遵守熵增定律，如果不去管理它，一定是从有序到无序，总有很多新需求发布，有很多新人进来，代码会增加，也会变得混乱。

一是减少熵增的速度，依赖良好的架构、分层、模块依赖关系等方面。

二是有性能防腐监控，能及时发现问题，减少对用户的影响，可以是开发阶段、ci/cd阶段，也可以是现网阶段，当然最好是在更前的位置发现和解决问题。

三是有清道夫角色长期进行清理。

3、考虑性能优化与开发/维护成本之间的平衡，很多性能优化的代码写法会增加维护成本，不仅是时间和人的成本，业务质量也会不可控，因为代码确实很难写，比如各种异步逻辑的处理，proxy代理的处理，最好是期望从框架层面去减少上层业务开发者的负担。

简单来说，当用户在地址栏敲下url并回车后，接下来的具体加载过程大致分为几部分：

- dns将目标域名解析成ip
- 对目标域名发起http建连

- 向目标域名请求并返回html
- 下载并解析html
- 构建dom节点
- 发起css, js等优先级高的请求
- 执行js, 并发起其他请求, 比如向后台获取数据
- 结合css渲染页面



可以抽象出这几部分优化点:

- 建连优化
- 动态资源加载优化
- 静态资源加载优化
- 必要数据获取优化
- 代码执行优化
- 业务流程优化

优化方案

建连

目标: 尽可能快完成站点建连

包括webserver和cdn的dns解析, http建连等

优化方法包括尽量**减少建连**和尽量**快速建连**

dns

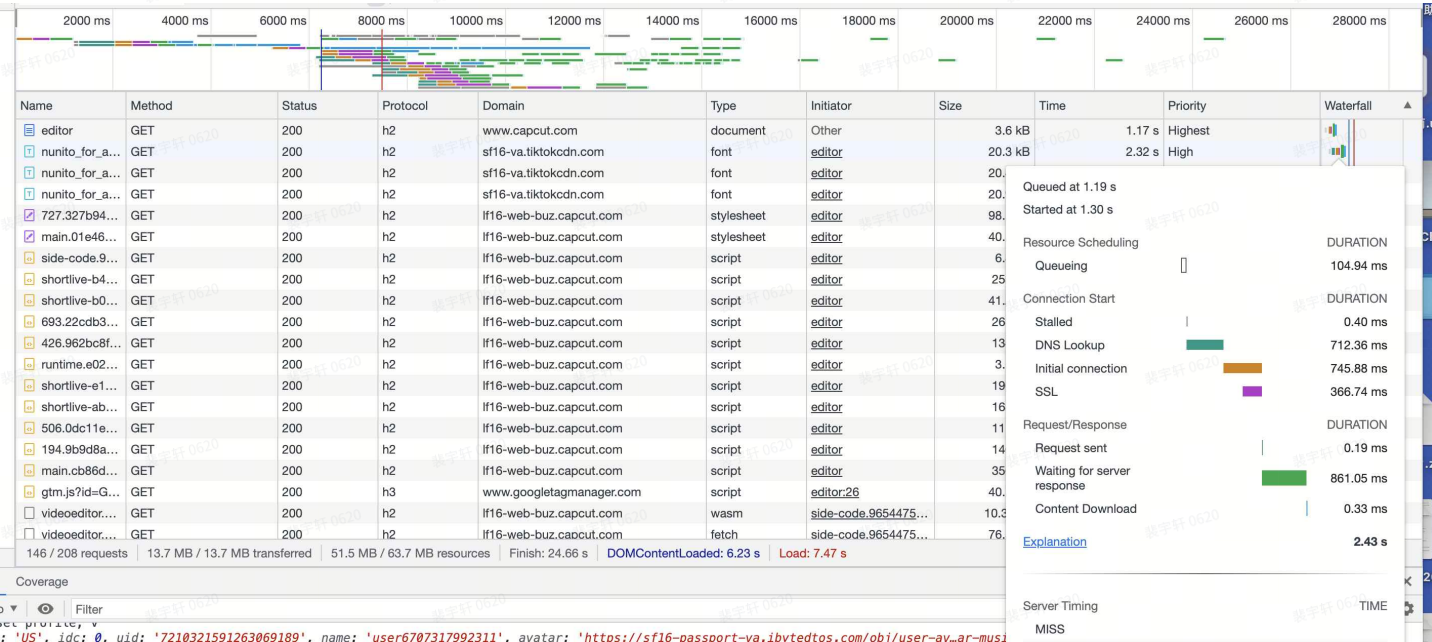
从浏览器缓存到运营商解析, dns最差可能得通过7层解析, 才能找到正确的ip地址, 这种情况下可能每个域名到ip的映射查找, 都需要50ms-200ms的时间, 优化方向主要分为2个: 减少dns解析和加快dns解析, 其中减少dns解析尤为重要, 且在日常开发中存在不少优化点

减少dns解析:

即站点用到的域名越少越好，特别是完整可编辑之前的域名，最优是保持2-3个：

- 1. 主站webserver域名
- 2. cdn域名
- 3. 备用cdn域名（可选）

目前cc-web的域名使用情况：



Name	Method	Status	Protocol	Domain	Type	Initiator	Size
collect?v=2...	POST	204	h2	www.google-analytics.com	ping	js?id=G-R9JUU6F3...	244 B
common-m...	GET	200	h2	sf16-short-va.bytedapm.com	script	426.962bc8f6.js:2	568 B
action.0.11...	GET	200	h2	sf16-short-va.bytedapm.com	script	426.962bc8f6.js:2	570 B
lv_web.ico	GET	200	h2	lf16-web-buz.capcut.com	vnd.microsoft...	Other	(disk cache)
user_info	POST + Preflight	200	h2	edit-api-va.capcut.com	xhr	426.962bc8f6.js:2	1.9 kB
webid	OPTIONS	200	h2	maliva-mcs.byteoversea.com	preflight	Preflight	0 B
captcha.js	GET	200	h2	lf-rc1.yhgfb-static.com	xhr	426.962bc8f6.js:2	(disk cache)
browser-set...	GET + Preflight	200	h2	mon-va.byteoversea.com	xhr	426.962bc8f6.js:2	1.4 kB
vendors~t.js	GET	200	h2	lf-rc1.yhgfb-static.com	script	VM71:3	(disk cache)
vendors~sld.js	GET	200	h2	lf-rc1.yhgfb-static.com	script	VM71:3	(disk cache)
get_user_w...	OPTIONS	204	h2	edit-api-va.capcut.com	preflight	Preflight	0 B
get_region	OPTIONS	204	h2	commerce-api-va.capcut.com	preflight	Preflight	0 B
bee_prod_4...	GET	200	h2	sf16-muse-va.ibytedtos.com	xhr	426.962bc8f6.js:2	228 B
abtest_config/	OPTIONS	200	http/1.1	vmweb-sg.byteoversea.com	preflight	Preflight	0 B
tobid	OPTIONS	200	h2	maliva-mcs.byteoversea.com	preflight	Preflight	0 B
tobid	OPTIONS	200	h2	maliva-mcs.byteoversea.com	preflight	Preflight	0 B
tobid	OPTIONS	200	h2	maliva-mcs.byteoversea.com	preflight	Preflight	0 B
storageExp...	GET	200	h2	lf16-web-buz.capcut.com	script	runtime.e02a256b.j...	35.3 kB
419d14d8d...	GET	200	h2	sf16-passport-va.ibytedtos.com	png	426.962bc8f6.js:2	(disk cache)
tiktok.b00c2...	GET	200	h2	lf16-web-buz.capcut.com	png	426.962bc8f6.js:2	82.8 kB

大概涉及了10多个以上域名，第一个图也可以看到dns-lookup就花掉了700ms，对于慢网速用户这里的消耗是非常大的。

结论：收归域名，特别是cdn资源尽量收归到1个域名上，现在都开启了http2之后，域名并发已经不是问题了，从cc-web出发，大致有这要做的事情：

- 严格控制新域名的数量，理论上最少2个域名就足够了
- 处理第三方js脚本使用的cdn域名，如上报库等，js执行是按顺序的，如果有个域名特别卡，有可能导致后面的js都执行不了，所以这里的时间一定要节省下来，通常如果不是需要更新非常及时的第三方库，都可以手动保存到自己域名上，宁愿麻烦一点手动更新，当然这里也有个额外的好处，即js内容更可控，避免因为外部js进行版本更新而导致自己网站白屏
- 字体文件放在别的cdn域名，这里要通过构建统一规范起来，不要随便引入域名
- cgi请求是跨域的，最好也收到主域里，但这里历史原因比较多，而且因为跨域还会多余发起一个perflight请求，这里也要处理

暂时更新不了的，如google的广告上报域名，不影响首屏的情况下可以暂时不管

加快dns解析

比如对dns进行预解析并加入浏览器缓存，现代浏览器都有dns-prefetch标签，在构建时扫描相关域名并且加入，也可以在前一级页面就提前进行dns解析，cdn域名的提前解析更重要，现代浏览器都比较智能，在浏览器地址栏敲入域名时一般就自动开始dns解析了。

也可以在html主页头部或者上级页面（如首页，工作台）加上dns预解析

http2

全部域名开启http2，不再需要雪碧图，也没有一个域名6个连接的限制，放心用

html加载

一般不会给html设置缓存，所以html页面的下载时间通常比同等大小的js资源要长一些，针对html页面的特点，有这些优化点：

压缩

记得给html开启压缩，包含里面内联的css

精简代码

不要有多余的代码、无用的标签和无用的标签属性、注释了未使用的代码等，能有多精简就有精简

流式下发

传统的html页面必须得全部下载完成之后才开始解析发起css、js等请求，或者构建dom，利用流式下发，可以加快这2个步骤，即边下发边解析执行，css和js请求可以在下发到这行代码时便开始请求。

特别是现在在server或者在网关处，会有很多服务运算逻辑，比如获取用户数据信息、草稿云解析等。这些时间都是算在ttfb时间里的，假设服务端的运算时间需要300ms，那么前端等待html的时间就至少是 请求往返时间 + 300ms服务端运算时间 + html文件的download时间，所有的css和js也都必须在这个时间后才能进行加载执行

如果使用html流式下发，server可以在服务运算好之前，就将主要的css和js先下发给浏览器，浏览器利用preload标签马上可以进行主要静态文件的加载，css和js的加载执行时间能大大提前，几乎仅仅等同于请求往返时间，不仅不用等待server端服务运算的时间，download时间都可以忽略不计了。

当然对于dom节点而言，也可以边下载边显示，对于ssr页面尤为有用。

http2 push:

http2 push可以用在这2个地方

- 1: 将css push下来，省去第一次从cdn建连和往返获取数据的时间，css加上dom节点并可以显示页面主要框架，特别适合用在ssr中，css的体积一般也比较小，不大会增加html的大小
- 2: 将首屏必需数据push下来，同样省去往返获取数据的时间，但因为目前协议和草稿内容的体积都比较大，效果和副作用待定

不要一次性传输过多数据

html页面从server下载的时候，一般会带一些必要信息，有时候为了减少前后端数据交互往返的时间，会将草稿等主要数据也直接从html页面里带过来。但过大的数据会导致ttfb和download时间都变得很长，所以要严格控制下发的数据量。

如果数据量比较大，可以仅先下发首片数据，或者用http2 push

ssr直出

ssr是提升首屏可见时间最有效的方法，即server端直接吐出不含用户交互的整个html页面，让用户尽量看到真实内容

cc-web的ssr需要解决这几个问题：

- 前端代码目前数据层和ui层是耦合在一起的，无法直接在server端执行并绘制出完整的dom结构，所以暂时采取暴力方法，将html页面的dom节点拷贝到server端吐出，后期等前端架构重构之后，再维护一份前后端同源的ui层
- 前端依赖后台的各个数据在网关处需要提前获取，随ssr一起吐出页面
- 草稿首帧同样由server端直出
- 需要注意不同状态的ssr页面，如登陆和未登陆等，防止页面跳动
- 由于前端js绘制出真正可交互的页面之后，需要干掉之前的ssr页面并进行重绘，会轻微影响可编辑时间
- 两个页面的绘制细节要同步好，避免用户觉得页面跳动



- 完整意义的ssr需要做到ui&逻辑分离，前后端共用一份ui生成的代码，代码同源
- 目前的ssr是写死dom来快速上线的，会有极大的维护成本，预计在架构调整时重构这一部分

静态资源加载

目标：尽可能快的下载各项静态资源，优先保证从缓存返回（memorycache -> pwa -> diskcache -> cdn -> 源站）

尽可能使用cdn

如js、css、图片、视频、字体文件等

缓存时间尽可能长

原先一些静态资源的缓存时间只有10分钟，达不到缓存的效果。现在缓存方案都是长缓存 + md5，可以放心的尽可能将缓存时间设置为更长，多消耗的资源可以忽略不计

cdn域名探测

cdn的网络质量偶尔会不稳定，或者根据地域的不同，使用不同cdn域名能获得更好的效果。我们可以在本地进行cdn拨测，找到效果最好的cdn域名，将结果存到cookie并跟构建关联起来，下次可以优先使用这个域名

开启更高压缩等级

目前资源压缩基本都是使用的br/gzip压缩，高等级压缩效果更好，解压多花费的时间又可以忽略不计，所以尽量把压缩等级开到最高，可惜的是目前我们的cdn暂时不支持自定义压缩等级

上级页面预加载

在很多hybird应用里，通常会由客户端去提前预加载webview页面的静态资源（如微信公众号），web应用没有客户端可以帮助预加载这些资源。但在某些条件下，可以在上级页面进行资源预加载，只要能满足这几个场景：

- 1. 二级页面的性能压力比较大
- 2. 一级页面性能压力比较小
- 3. 一级页面跳往二级的页面的频率比较高

我们的首页 -> 编辑页，工作台 -> 编辑页 都满足这几个场景

具体做法有这几种，在一级页面里：

	优点	缺点	
prefetch	<ul style="list-style-type: none">1. 简单2. 发起时间过晚，不影响一级页面的性能	<ul style="list-style-type: none">1. 发起时间晚，通常来不及缓存二级页面的内容，用户已经跳转	
preload	<ul style="list-style-type: none">1. 简单	<ul style="list-style-type: none">1. 发起时间过早，有可能影响一级页面的性能	
iframe	<ul style="list-style-type: none">1. 发起时间可控2. 短时间内跳转，资源可进入memorycache，读取速度更快	<ul style="list-style-type: none">1. 占用大量资源，包括动态请求等2. 二级页面重复执行上报	
手动插入	<ul style="list-style-type: none">1. 发起时间可控2. 加载资源可控	<ul style="list-style-type: none">1. 实现相对复杂2. 有接入成本	

最终剪映-web还是选择最后一种手动插入的方式，虽然有一些实现和接入成本，但发起时间和加载资源数量可以手动控制，有机会在上级页面资源占用和下级页面资源缓存率之间寻找一个相对平衡点

实现方案：

通过构建将下级页面所需要的首屏相关资源包打成json文件，发布到cdn，上级页面加载这个文件，并在自己合适的时机对json文件里的资源发起preload缓存请求

注意点：

- 需要监控对上级页面的性能影响
- 需要监控和调整下级页面的资源缓存成功率，如果偏低，考虑将缓存发起时间再提前，直至寻找到平衡点

资源包大小平均

http2能保证相同优先级的静态资源在差不多时间发起，但结束时间通常会跟最大的包相关，如原来有1.2m大小的js资源包，会整体拖慢整个静态资源的下载，需要保证每个包大小基本一致

根据之前的测试结果，每个包在50k到100k能达到较好的加载效果。

Name	Status	Protocol	Domain	Type	Initiator	Size	Time	P..	Waterfal
side-cod...	200	h2	lf16-web-buz.capcut.com	script	editor	1.8 kB	3.81 s	H..	
main-f88...	200	h2	lf16-web-buz.capcut.com	script	editor	51.9 kB	4.18 s	H..	
runtime....	200	h2	lf16-web-buz.capcut.com	script	editor	3.1 kB	3.81 s	H..	
thirdpart...	200	h2	lf16-web-buz.capcut.com	script	editor	85.1 kB	2.37 s	H..	
213.0f9f...	200	h2	lf16-web-buz.capcut.com	script	editor	40.2 kB	4.14 s	H..	
274.241...	200	h2	lf16-web-buz.capcut.com	script	editor	55.4 kB	4.13 s	H..	
byted_im...	200	h2	lf16-web-buz.capcut.com	script	editor	50.8 kB	4.12 s	H..	
byted_c...	200	h2	lf16-web-buz.capcut.com	script	editor	64.9 kB	4.04 s	H..	
themes....	200	h2	lf16-web-buz.capcut.com	script	editor	34.1 kB	2.95 s	H..	
930.641...	200	h2	lf16-web-buz.capcut.com	script	editor	42.1 kB	4.13 s	H..	
158.b9d...	200	h2	lf16-web-buz.capcut.com	script	editor	58.4 kB	4.03 s	H..	
main-c3...	200	h2	lf16-web-buz.capcut.com	script	editor	48.5 kB	4.18 s	H..	
main-e2f...	200	h2	lf16-web-buz.capcut.com	script	editor	43.7 kB	4.00 s	H..	
main-93...	200	h2	lf16-web-buz.capcut.com	script	editor	42.2 kB	4.13 s	H..	
main-c5...	200	h2	lf16-web-buz.capcut.com	script	editor	35.8 kB	3.79 s	H..	
main-8c...	200	h2	lf16-web-buz.capcut.com	script	editor	41.7 kB	4.14 s	H..	
main-31...	200	h2	lf16-web-buz.capcut.com	script	editor	64.6 kB	4.17 s	H..	
thirdpart...	200	h2	lf16-web-buz.capcut.com	script	editor	17.9 kB	3.81 s	H..	
699.0c9...	200	h2	lf16-web-buz.capcut.com	script	editor	52.1 kB	4.11 s	H..	
byted_im...	200	h2	lf16-web-buz.capcut.com	script	editor	46.3 kB	4.12 s	H..	
byted_im...	200	h2	lf16-web-buz.capcut.com	script	editor	69.4 kB	4.02 s	H..	
byted_im...	200	h2	lf16-web-buz.capcut.com	script	editor	58.7 kB	4.04 s	H..	
byted_im...	200	h2	lf16-web-buz.capcut.com	script	editor	29.1 kB	3.79 s	H..	
byted_c...	200	h2	lf16-web-buz.capcut.com	script	editor	66.5 kB	4.05 s	H..	

目前剪映web的js资源加载总体比较平滑，可以看到几乎能在同一时间发起和结束。

（仅对首屏资源做处理即可，非首屏资源反而不用，它们根据模块关系来各自打包可以更好的提高发布之后的缓存命中率）

保证静态资源优先级合理

这个优化点很容易被开发同学忽略，网络带宽总是有限的（目前cdn拨测的结果大概是2m/s-10m/s，平均4m/s），我们需要把有限的网络带宽优先留给更加重要的资源

从v8代码里可以找到资源优先级：[从Chrome源码看浏览器如何加载资源](#)

```

ResourceLoadPriority TypeToPriority(Resource::Type type) {
    switch (type) {
        case Resource::kMainResource:
        case Resource::kCSSStyleSheet:
        case Resource::kFont:
            // Also parser-blocking scripts (set explicitly in loadPriority)
            return kResourceLoadPriorityVeryHigh;
        case Resource::kXSLStyleSheet:
            DCHECK(RuntimeEnabledFeatures::XSLTEnabled());
        case Resource::kRaw:
        case Resource::kImportResource:
        case Resource::kScript:
            // Also visible resources/images (set explicitly in loadPriority)
            return kResourceLoadPriorityHigh;
        case Resource::kManifest:
        case Resource::kMock:
            // Also late-body scripts discovered by the preload scanner (set
            // explicitly in loadPriority)
            return kResourceLoadPriorityMedium;
        case Resource::kImage:
        case Resource::kTextTrack:
        case Resource::kMedia:
        case Resource::kSVGDocument:
            // Also async scripts (set explicitly in loadPriority)
            return kResourceLoadPriorityLow;
        case Resource::kLinkPrefetch:
            return kResourceLoadPriorityVeryLow;
    }

    return kResourceLoadPriorityUnresolved;
}

```

可以看到优先级总共分为五级：very-high、high、medium、low、very-low，其中

可以对应这些规则来调整资源优先级：

1. 跟首屏渲染相关的css、js需要保证是第一优先级，不能使用动态请求的方式
2. 必要server相关数据请求要保证是第一优先级
3. 可交互相相关的css、js是第二优先级
4. 比较少用的功能相关css、js资源优先级靠后
5. 其他可以延迟的资源尽量不要抢占网络，如上报，不重要的数据获取等

一般我们优先在架构上（高性能分层/单向依赖架构）来保证资源能够被合理加载，另一方面，也可以简单的在控制台逐个排查不合理情况并加以修正，目前剪映web已经全部修正一轮，有不错的结果

无用代码去除

因为往往会有很多不需要的代码被打包进项目，越大的项目，开发人员越多，周期越久的项目里这个问题越严重，比如ccweb和一些比较大型业务的首屏文件的无用代码都达到了60%以上。

URL	Type	Total Bytes	Unused Bytes	Usage Visualization
https://sf3-scmcdn2-cn.feishucdn.com/eesz/bear/docx/module/ee/do.../index_merged_es6.js	JS (per ...	4 895 382	3 009 506	61.5%
https://sf3-scmcdn2-cn.feishucdn.com/ccm/pc/.../docx_app_next.335606e2e4e799a21a06.js	JS (per ...	4 777 341	2 202 718	46.1%
https://sf3-scmcdn2-cn.feishucdn.com/.../bear-docx-loadable-bidirection.1b787af65df8e5df154b.js	JS (per ...	2 203 899	1 795 060	81.4%
https://sf3-scmcdn2-cn.feishucdn.com/ccm/pc/web/reso.../36796.ce71f21e74d2778d293d.js	JS (per ...	1 758 362	1 750 089	99.5%
https://sf3-scmcdn2-cn.feishucdn.com/ccm/pc/web/reso.../40921.4deaaae0b8d27b7e15a7.js	JS (per ...	1 711 872	1 709 505	99.9%
https://sf3-scmcdn2-cn.feishucdn.com/eesz/bea.../docx_abbreviation_es6.b692214b.chunk.js	JS (per ...	2 275 526	1 429 252	62.8%
https://sf3-scmcdn2-cn.feishucdn.com/eesz/bear/doc.../docx_toolbox_es6.2df48a30.chunk.js	JS (per ...	1 511 236	1 427 566	94.5%
https://sf3-scmcdn2-cn.feishucdn.com/ccm/pc/web/reso.../76780.8b8e90355f5be4861321.js	JS (per ...	1 879 731	1 377 527	73.3%
https://sf3-scmcdn2-cn.feishucdn.com/ccm/pc/web/reso.../32054.75fb36c969e2a280b5d4.js	JS (per ...	1 854 046	1 315 110	70.9%
https://sf3-scmcdn2-cn.feishucdn.com/ccm/pc/.../diagram-editor.dd4e5126a478d60628c5.js	JS (per ...	1 759 730	1 274 924	72.4%
https://sf3-scmcdn2-cn.feishucdn.com/eesz/bear.../docx_index_delay_es6.c404f673.chunk.js	JS (per ...	1 385 103	1 105 391	79.8%
https://sf3-scmcdn2-cn.feishucdn.com/ccm/p.../doc_index_css.c4c7d931dc65868a2292.css	CSS	1 114 881	1 060 968	95.2%
https://sf3-scmcdn2-cn.feishucdn.com/ccm/pc/web/reso.../86175.fe3abd8ae509eab71ea9.js	JS (per ...	1 182 179	1 022 059	86.5%
https://sf3-scmcdn2-cn.feishucdn.com/ccm/pc/web/resou.../54948.f37b5498343525ef7e5b.js	JS (per ...	988 156	966 723	97.8%

优化方案：

1. tree-sharking能分析并去除一部分没有被依赖或者使用到的代码
2. 对于被import进来，也确实被其他模块依赖、但是首屏不需要的模块，webpack有工具可以分析全部的依赖关系，然后可以手动从模块上逐个排查其被依赖的合理性
3. 模块内部级别上，因为类/函数不够内聚，违反单一职责原则，一个文件里总会有很多本不应该存在的代码，chrome控制台有代码覆盖率检查工具，可以逐行检查代码是否真正需要，通过这种简单粗暴的方式，剪映-web首屏减少了大约25%的js代码

注意点：

对无用代码的清理永远不可能做到100%完美，且很容易被腐坏，在优化过程中建议抓大放小，专注于roi更高的地方

重复代码去除

重复代码主要出现在这些场景，如：

同一个npm包被重复引用

外部sdk引用了重复代码

复制粘贴了大量相同代码

主要解决方案：

1. 大仓，减少模块重复互相引用的概率
2. webpack external
3. 禁止复制粘贴重复代码
4. 写脚本工具分析

减少资源变动概率

每次发布时，一旦某个文件里有任何改动，用户便需要去完全重新下载整个文件，我们的目标是尽可能让用户持续使用缓存里的文件，而不是每次都需从服务器上重新下载。虽然需求更新肯定会导致对应的代码更新，但我们可以给资源文件分级

- 从变动频率上，把不常变动的文件打包在一起，经常变动的文件打包在一起
- 从业务逻辑上，同一个功能的相关文件尽量内聚打包在一起，一次需求发布影响的文件尽可能少，从而提高缓存效率。要做好这一点，也需要有合理的模块依赖关系，以及良好的单向依赖分层架构。我们希望做到，当需求变更时，随之需要变更的代码尽可能少。


动态资源转化成静态资源加载

静态资源通常都是通过cdn加载，比起webserver上的资源，加载速度是有很大优势的。每当我们看到一个资源通过http接口等形式动态加载的时候，都可以想想它是否可以变成静态资源来进行加载。

一个很典型的例子就是多语言，之前多语言是通过http接口去动态获取语言包，大盘数据大概花费800秒，这个流程明显是不合理的，多语言是通过静态编译就可以决定资源和逻辑，不需要等到动态获取阶段。

在代码构建编译阶段就应该将多语言打成不同的html->js制品，由cookie信息跟html制品在网关处关联起来，用户在哪个国家，就对应进入不同语言的html制品。

以此类推，我们要牢记这两个原则



能在编译阶段决定的逻辑，就不要等到动态运行时

能用cdn加载的资源，就不要放在动态服务器上

相同优先级的资源并行加载

不管页面里有多少静态资源，最理想的加载方式是它们能同一时间开始发起请求，并且在同一时间结束请求。

- 同一时间结束请求依赖包大小的分配情况，当然也没有办法保证有100%最理想的结果。
- 同一时间开始请求也不是很现实，网络带宽总是有限的，最好的办法就是保证相同优先级的资源并行加载，比如首屏相关的资源同一时间发起，部分可编辑相关资源同一时间发起，全部可编辑相关资源同一时间发起。
- 重要的静态资源千万不能出现串行加载的情况。

合理使用各种加载方式

- 目前大部分功能模块都是立即加载的，从产品体验出发，应该更合理的使用各种加载方式，如：
- 立即加载
 - 闲时加载
 - worker加载
 - 点击加载
 - 按需加载（按场景加载）

从性能出发，上述越后面的加载方式对页面的打开性能影响越小，但也有一些各自的缺点

	优点	缺点	适用模块	
立即加载	1. 代码简单可控	大量资源第一时间下载，性能差	大部份首屏必需模块，或者用户第一时间会使用的模块	
闲时加载	1. 代码简单 2. 不影响首屏性能	闲时队列不好控制，可能造成用户操作时卡顿	用户第一时间不会使用的模块，如大部份上报代码，一	

			些低频使用的组件	
worker加载	不阻塞主线程	<ol style="list-style-type: none"> 1. 只适合纯逻辑模块 2. 跟主线程通信比较麻烦 3. 跟主线程通信可能耗时大 4. 无法第一时间被初始化，所以不适合第一时间需要使用的模块 	函数计算 编解码等	
点击加载	性能好	<ol style="list-style-type: none"> 1. 需要额外的代码来控制逻辑 2. 无法离线编辑 	大部分依赖用户操作才执行的模块	
按需加载	性能好	需要额外的代码来控制逻辑	如非登录场景下不需要显示用户信息模块等	

要做好模块的加载总体来说十分不容易，总是无法避免这几个问题

1. 需要根据产品逻辑和上报数据给每个模块划分最合理的加载方式
2. 代码难写，每种加载方式的代码写法都不一样，同步和异步混用，上层业务开发者压力大
3. 新人理解成本高
4. 在需求迭代过程中很容易腐坏

但也可以通过合理的架构来减轻这些问题带来的负担，后续的架构篇会细讲。

巨型静态资源特殊优化

wasm sdk是目前剪映-web最大的静态资源，目前大概共有30m大小（gzip前），按照当前在北美对cdn的拨测速度推测，理想状态下平均也要2.5-3s来下载，当前缓存命中率为60%左右。但是几乎每次发版后用户都需要重新下载这个巨大的资源包。而且最难受的是这个sdk几乎是首屏和可交互之前的必需资源，连延迟加载都没法做。

所以我们必须尽可能提高它的缓存命中率，当前最影响缓存命中率的显然是每次发版，wasm的md5都会变化。可以借鉴web离线包的方式，我们会在本地长期保存2个以上的版本，然后在页面打开时总是先使用本地已有的wasm包，直到在空闲时间或者上级页面加载最新的资源包，保证用户尽量从缓存里获取资源。

这个方案的缺点就是需要一套合理的版本更新机制，且用户可能要多刷新一次才能使用最新的wasm资源包，但对于该方案能带来的性能提升来说，整体还是很划算的。

使用代理加载

代理模式在一些性能优化场景也很有用，总体思想是先加载一个更小的模块，这个模块对外暴露的api跟本体模块是保持一致的，只有在真正需要本体模块被加载的时候，才会去加载真实所需要的代码，这个过程过于上层业务开发者来说是透明的。

举个上报模块的例子：

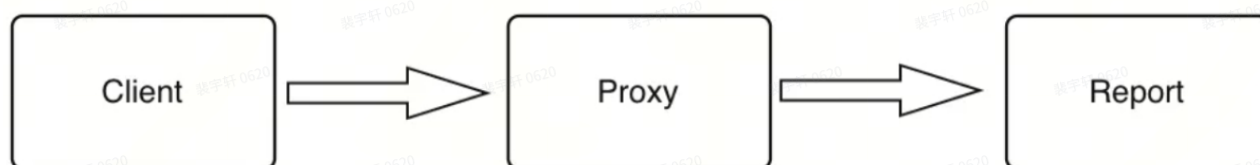
我们都知道，上报是非常基础的功能，站在上层业务开发者的角度，通常我们会将通用上报库在html的最前面就加载进来，因为可能需要在任何时机发起上报动作。

但上报操作其实也是相当耗资源的，一方面公司的上报系统很多，产品和开发需要观测的数据也很繁杂。导致整个上报基础库非常庞大，占用的下载资源会影响其他重要模块，比如可见和可编辑模块的加载。另一方面，这些上报动作也会抢占http上行带宽。

其实我们对具体上报的时机没有太高要求，服务端早几秒和晚几秒拿到上报数据没有区别。我们最好可以让上报代码加载和上报动作的执行都在浏览器空闲的时候去执行，将最宝贵的时间留给首屏元素渲染。

我们可以创建一个跟report仓库具有相同的接口的proxy类，强制业务代码统一去调用proxy类，proxy类并不包含真正的上报代码，当Client开始上报动作时: Proxy只需要做这两件事情：

- 如果Report本体已经加载，直接将请求传递给Report本体
- 如果Report本体还没有加载，则将请求保存起来，等Report本地加载完之后再请求传递给Report本体



Proxy类的代码只有几十行，通过这种方式，我们可以解决加载Report库和具体Report代码带来的性能问题。

使用proxy的方式，除了上报模块之外，还有很多其他同步模块可以用类似的方式，转化为异步去加载执行。但有2个重点的地方需要注意：



- 可以将同步模块转化为异步去加载执行的前提是，要么其他模块不关心这个模块的处理结果，要么就需要处理好同步模块转化为异步模块之后，这些模块和其他模块的依赖关系
- proxy要和本体拥有一样的接口，上层业务开发者不用去理解他们在接口背后的真正差异。作为消费者，不愿意去关注它调用的到底是不是proxy，不能因为使用了proxy而给上层业务开发者增加额外的负担，否则可能得不偿失。在上层业务开发者看来，他调用的就是原来的模块。

一些检查方法



chrome控制台就提供了非常便利的检查方法，通过这些方式我们很好的优化了相关资源加载性能

- 检查代码覆盖率
- 检查资源&请求优先级
- 检查缓存时间和缓存方式
- 检查http2
- 检查域名情况
- 检查包大小情况
- 检查资源合理性
- 检查异步资源是否需要同步
- 检查优先级为低的js资源是否合理

数据获取

目标：尽可能少次数与server进行数据交互，必要交互尽可能快



那么抽象出来的思想有：

- 减少交互
- 交互提前

- 交互并行
- 分批获取数据
- 追溯慢的基本原因

减少数据交互

数据获取耗时是因为用户的终端需要跟后台产生数据交互，既然如此，最好的解决办法就是前后端之间少产生或者不产生数据交互

当然前后端之间完全不产生数据交互是不可能的，我们可以让关键数据在第一个打开页面时并返回，避免前端向后端重新发起请求，比如登录信息就完全可以在网关注入，实际上剪映web将原本在前端发起的3次数据调用都挪到后端，能大幅减少数据交互的时间

草稿和轨道等协议信息也最好由server直接返回，免去前端重新请求的时间，但目前草稿等协议信息还不支持分片，很多时候资源过大反而会影响首屏性能，所以暂时不做处理

也有一些非关键数据可以存在前端本地缓存里，如localstory或者indexdb，避免跟后端产生频繁交互

数据交互提前

如果前后端必须进行数据交换，那就最好让这些数据交互在最早的时机发起，即页面打开的第一时间。

但普通http请求的优先级为low，无论代码写的多靠前，只要是在代码里发起的http请求，发起时间都会比较靠后，至少在绝大部份静态资源加载之后，唯一的办法是用把这些请求变成script请求，即用jsonp方式来发起这2个请求。

目前我们的草稿协议接口就非常适合用jsonp方式，上线后也有不错的优化结果

注意点：



需要后端支持callback方式返回jsonp数据，且请求所需参数需要用构建在静态编译时就能获取，再打进html里，通过script标签内容的后缀传给server

数据交互并行

如果前后端需要两次以上的数据交互，除非这几次交互有逻辑上的先后强依赖关系，那就应该让这些请求全部并行发出，千万不要进行串行等待

减少数据转换

预期server返回的数据，前端不需要经过太多处理就可以直接使用，比如目前server返回的草稿是一个多层嵌套结果的json字符串，前端需要花上几十-几百ms来进行parse，更好的方式是server直接返回一个object，前端无需parse即可使用

对于server返回的前端信息，一般前端需要转换为自己的数据结构才更方便使用，所以也希望server返回的json object能尽量和前端数据结构模型接近，减少二次转换的过程

分批获取数据

对于数据量大的请求，通常一次请求的ttfb时间、download时间、以及拿到数据后在前端进行渲染绘制的时间都很长，特别对于在线文档、在线编辑器这样的业务而言。

比如200万个单元格的在线表格、复杂草稿的视频编辑器，这些业务场景下前后端需要交互的数据量一定是非常大的，唯一的优化方式就是分批获取并且分批渲染数据

cc-web可以从这几个纬度来分批获取数据

- 草稿/轨道协议时间轴上分片获取数据
- 草稿分层获取数据
- 多轨道按轨道分批获取数据

理论上通过分批获取数据的方式可以大大降低复杂草稿的上屏和操作时间，但也有一些副作用，需要额外关注：

- 代码逻辑会复杂很多，最好有统一的资源加载器来减轻上层业务开发者的精力负担
- 总的完整可编辑时间会延长，需要和单次编辑时间之间寻找一个平衡点

防止数据交互影响其他代码逻辑执行

在第一批性能优化过程中，就发现一个比较低级的问题，多语言数据包是通过http接口获取的，因为后面的ui渲染会依赖多语言信息，所以整个过程是串行的，只有等多语言数据都拉下来之后，才会继续执行后面的代码。

拉取多语言数据包的大盘时间是800ms左右，这显然是很不合理的。我们要避免数据交互会串行的影响其他代码逻辑执行。

方法主要有这几个：

- 将数据获取放置于server，再吐给前端，前端可以直接使用
- 将数据获取时间压缩，并和前端代码执行尽量并行，比如用jsonp发起草稿信息等

总而言之，虽然前端代码逻辑有时候难以避免的需要依赖某些前后端数据交互，但我们要想办法压缩和并行这个过程，将影响降至最小

追溯慢的基本原因

前后端交互慢的根本原因需要逐个排查解决，比如服务接口合并，服务接口优化，网络优化等

代码执行

虽然现在引擎对js代码执行做了各种各样的优化，但总体来说js的执行性能并不高，且由于js没有真正的多线程（worker缺点还比较多），太多或者太复杂的代码执行经常会卡死js主线程。所以js的执行效率和性能特别重要

首屏相关任务第一时间阻塞式执行完

不管用什么手段优化，总有很多代码是需要在首屏的第一时间内执行完的，比如基础库、页面上的主要首屏组件渲染、应用的启动器等。这些任务执行完的时间决定了页面什么时候能开始显示。

我们记住这几个原则：

- 首屏必需的任务越少越好，非首屏需要的任务都尽量用各种延迟加载/按需加载/闲时加载/server执行的方式往后面丢
- 首屏必需的任务第一时间阻塞式密集执行完，尽量让浏览器空闲时间早点到来，用户对这段等待时间也有一定预期
- 这个过程必须控制在500ms以内，最好是200ms-300ms

尽可能少的执行代码

web性能优化的关键其实就是尽可能少的加载代码和尽可能少的执行代码，代码加载和代码执行往往是强关联的，加载的代码更少，理论上需要执行的代码就更少。

所以如何减少代码执行，主要实际就是看如何减少代码加载，相关具体优化方法可以参考静态资源加载优化部分。

除此之外，也有一些可以注意的优化点：

- **减少自执行代码**

如果我们平时有分析页面的代码执行情况，大概会发现有30%-50%的时间都花在了自执行代码上，其中又可能只有一小部分代码逻辑是真正需要在页面打开之前自执行的。优化原则是严格控制自执行代码行数，所有自执行代码理论上都可以被封装成task，再由应用启动器决定是否要真正执行

- **代理**

- **防抖和节流**

原理和作用不用细讲，这里一个比较容易踩的坑是一些影响性能的全局事件的滥用，比如window.resize，首先window.resize肯定是要做节流的，其次这些全局事件也要被统一管理起来，不能让上层业务开发者随意去增加一个window.resize事件，每个频繁触发的window.resize都会严重影响性能，而是应该由基础底层模块来提供统一对外的事件

- **Currying执行代码**

部分密集计算的代码可以用currying化的原理来延迟计算过程，网上有案例，这里不再赘述

不要用定时器控制代码流程

我在几个比较复杂的业务里都见过用定时器来控制代码流程，一是可能因为逻辑流程复杂，二是很多时候不同场景下的逻辑流程又存在不一样的情况，为了简单省事，就设置了xxx毫秒之后从流程x进入流程y

这样的代码可能导致2个问题：

- 本身逻辑存在漏洞，定时器不能百分百保证流程跳转是正确的
- 定时器来控制代码流程，特别是应用的启动流程，会存在代码空转的情况，即很多可能在下次定时器触之前，是没有需要被执行的代码被真正执行的。

对于整个系统的主要逻辑流程，特别是应用启动流程，一定是要从根本上梳理清楚的，可以用task + 应用启动器来管理整体的启动流程

善用闲时执行

利用requestIdleCallback，我们可以找到一些浏览器的空闲时间，有许多任务都可以在这些空闲时间执行，而不是跟更重要的任务去抢占ttv或者tti时间

比如上报、大部份不重要的组件、对下级页面的资源预加载等任务

注意点：



如果滥用闲时执行不加以管理，可能造成一种情况，闲时队列里塞满了太多长任务，反而导致用户的操作卡顿

要解决这个问题，可能需要对闲时执行做一些管理，比如：

对长任务做好切分，不要有longtask

对任务分级，按重要程度等分好优先级

建立并管理好闲时任务队列，优先级更高的任务、或者用户主动触发的任务可以插队执行

保证异步模块和同步模块/其他异步模块之间的正确逻辑依赖关系

具体可以参考angular zone.js或者react fiber对闲时任务的一些处理思想

合理使用worker多线程

合理使用worker可以很大程度上减少对主线程的压力，但目前worker还有很多场景不太适用，开发原则是尽量把代码往worker里挪，如果能提前评估好这些问题。

- 对于上层业务开发者而言，使用worker还是有些额外成本的，目前没有太好的方式完全抹平上层业务开发操作主线程和worker的差异，这部分开发&维护成本在使用worker的时候要尤为注意
- worker线程只能在主线程和主域下动态发起，发起时间不可能特别早，所以一般在worker里运行的模块都不能是首屏强依赖的，否则会影响首屏的打开时间
- worker目前只适合跑纯逻辑层代码，所以模块的ui&逻辑分离要先做好
- worker跟主线程有一定的通信成本，只能通过字符串通信，也无法共享内存，如果数据过大，通信可能都要几百毫秒以上，需要和worker带来的性能收益进行对比
- worker会带来额外的内存开销，也需要严格控制worker的内存占用以及总的worker数量
- worker线程内部也可能产生拥挤，最好有个统一的管理器来合理管理worker的创建、销毁，保持合理的worker数量

耗时更多的重要任务尽早异步执行

首先我们希望很多执行耗时比较大的任务是可以异步执行的，这个异步可以发生在worker里，也可以交给后台，总之尽量不要跟主线程抢占宝贵时间，比如草稿预解析、worker初始化wasm等，都遵守了这个思想。

同时对于这些耗时比较大的异步任务，如果它发起的时间过晚，往往别的代码都全部执行完了，还需要继续等待这个任务的执行。

所以尽量让这些任务在应用初始化后第一时间发起，减少应用总的初始化等待时间。

proxy延迟执行

这是个很有用的技巧，而且使用成本很低。方案最初来源于vscode的依赖注入模块。

vscode的依赖注入代码提供了对象延迟初始化能力，这个能力虽然实现简单，但是给业务带来的实际帮助效果非常不错。

系统中有一些对象，比如工具栏的undoredo模块，为了便于提供给其他模块使用，我们一般会在程序的启动阶段就初始化好一个undoredo对象。

如：

```
1  class app{
2      undoRedoStack: UndoRedoStack
3      constructor(){
4          this.undoRedoStack = new UndoRedoStack;
5      }
6  }
```

这样对于上层应用开发者是比较友好的，他们可以随时使用undoRedoStack对象：

```
1  app.undoRedoStack.push(element);
```

但实际上，在用户真正产生操作数据之前，用户并不需要使用undoredo栈，所以这时候是不需要提前来创建一个undoredo对象的。一方面，太早去创建这个对象，消耗的时间会影响其他更重要代码的执行。另一方面，如果用户一直没有编辑数据，那这个提前创建好的undoredo栈对象可能就白白浪费了内存。

一般情况下，我们可以采取一些办法，来刻意控制undoRedoStack的创建时机。

如：

```
1 class App{
2     undoRedoStack: UndoRedoStack
3     constructor(){
4
5     }
6     getUndoRedoStack(){
7         if (this.undoRedoStack){
8             return this.undoRedoStack;
9         }
10        return this.undoRedoStack = new UndoRedoStack;
11    }
12 }
```

当消费者真正开始使用undoRedoStack时，需要通过app的getUndoRedoStack方法：

```
1 app.getUndoRedoStack().push(element);
```

我们专门为消费者提供一个getUndoRedoStack函数，来控制undoRedoStack对象的生成时机，以及undoRedoStack成为单例对象。

但这种方式是有代价的，代价就是系统中白白多了getUndoRedoStack这个方法，开发者要阅读、理解和维护getUndoRedoStack方法，这些都是要持续付出的成本。

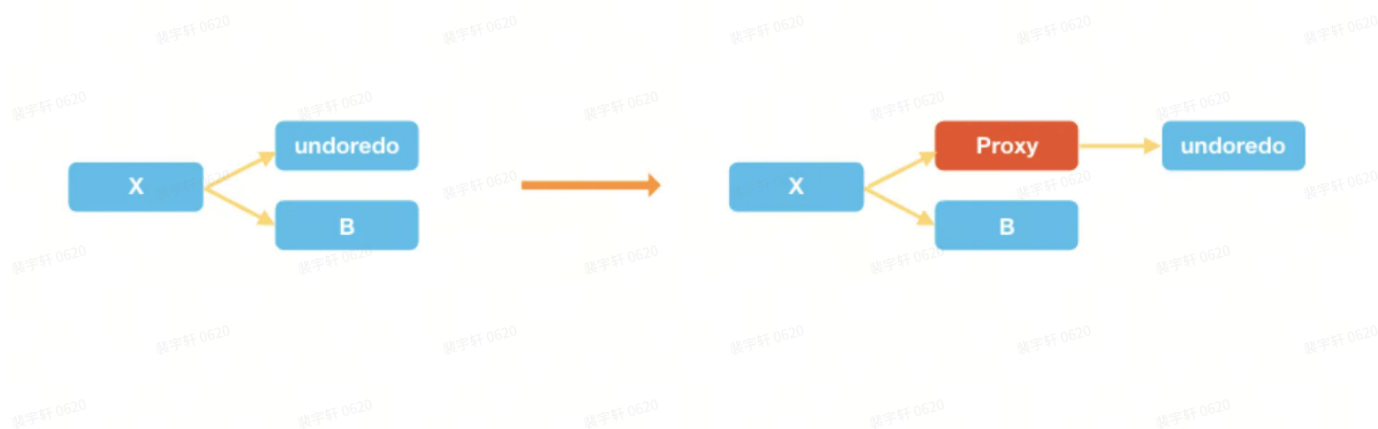
如果系统中有许多对象需要用这种方式来完成延迟创建对象的能力，那我们的系统将充斥着这些和核心业务逻辑无关的工厂方法。

vscode结合依赖注入容器，用一种简单而巧妙的方式，将对象的初始化时机延迟到消费者真正需要使用这个对象的时候。

具体思路如下：

当vscode的依赖注入容器分析完class与class之间的依赖关系之后，并不会真正去创建一个真实对象实例，而是先创建一些没有任何属性的空对象作为proxy代理来占位，这个空对象占用的资源是很少

的，它占用掉的内存和创建它所花费的时间都可以忽略不计。



接下来需要劫持这些proxy代理的get方法，当消费者真正开始调用它想要调用的对象的时候，这时候才会在proxy的get方法里去new出对应的真实对象，消费者的每一次调用实际上都传递给了proxy，proxy再委托给真实对象。

伪代码：

```
1 const undoRedoProxy = new Proxy(Object.create(null), {
2   get(target, key){
3     // 没有实例时，先创建实例
4     if(!this.instance){
5       this.instance = di.create(Ctor);
6     }
7     // 返回已经缓存的实例
8     return this.instance[key];
9   }
10 });
```

这样一来，业务逻辑里就不用再关心这些对象的创建时机，它们会在真正需要的时候才被创建出来，用这种proxy代理的方式，既提高了页面性能，又能保证核心业务逻辑不受污染，对业务开发者也是完全透明的。

实际代码逻辑里，按照以上方案，可以有大量的对象被proxy托管，对整体性能可以产生很正面的效果

防止主线程空转

经常会因为有定时器之类，导致主线程产生等待（空转），可以简单的通过排查chrome performance发现这些问题

切分主线程longtask

因为js的单线程机制，一旦主线程运行繁忙的时候，就会导致用户操作卡顿，通常这种现象都是longtask造成的，即用户感官内的一帧（16ms左右）无法运行完当前js任务

通常如果逻辑里存在longtask，且这个longtask是合理的，那么无法从源头避免longtask的产生。但我们可以尝试把longtask拆成更小的一系列task，然后以1帧为间隔（16ms左右）来运行这些小task，中间可以适当用定时器来隔断任务的连续执行，保证浏览器在用户感官的每一帧后，都能留给用户一些主动操作的时间和机会。

这种方案的缺点一是有额外的开发和维护成本，二是会增加总的任务完成时间，所以一般不适合首屏强相关的任务执行。

业务流程优化

目标：在尽可能满足用户体验的同时，对业务流程进行优化调整，在最简单的逻辑流程下，能完整渲染符合用户预期的页面

按照具体场景设计更合理的加载流程链路

假设页面有10个组件，完整可编辑页面依赖这10个组件的全部加载完成，那么合理的加载顺序是否一定是1、2、3、4、5，直到10呢？

答案显然不是，在不同场景下，这些组件的不同加载顺序经常能组合成更好的结果

比如剪映-web空草稿页，就跟草稿页的加载顺序不一样。草稿页需要第一时间显示草稿内容，所以必须第一时间加载wasm sdk。

而在空草稿页，用户第一时间是没有草稿使用的，所以用户需要第一时间加载云相册sdk来进行上传操作。空草稿页在用户体感上的加载速度也会快很多，因为不用第一时间加载30m大小的wasm sdk，而当用户完成上传操作需要显示草稿时，wasm sdk往往已经被加载好了。

在不同场景下，灵活来组装模块的加载顺序可以取的很好的收益，一般可以这样进行优化：



- 提前设计一个好用的应用加载管理器
- 按场景画出模块依赖图/加载链路图，排查每种场景下的加载链路并进行分析和优化

按照用户操作习惯设计更合理的加载流程链路

页面中每个组件被用户真正使用的概率和频率是不一样的，草稿、轨道、上传正常来说就比分享按钮使用的概率更大一些。我们可以上报每个组件的使用情况，将更有可能被使用的组件放在更前面的位置进行加载。

这种方案也有开发&维护成本的问题，也可能需要经常去调整组件加载顺序，所以需要提前设计一个可以配置化的组件加载系统。

草稿协议精简

中间层草稿协议较，复杂，包含了许多web端不需要的字段。最好是在源头对web端的中间层协议进行精简，但周期比较长，前期考虑web端用一份精简的默认配置进行下发

草稿分片&协议分片

背景：大草稿数据量很大，加载和渲染极慢，影响业务上限

优化方案：像文档一样进行内容的分片加载和渲染

草稿和轨道分层拉取和上屏

背景：当草稿和轨道资源比较大时，拉取数据和上屏都很慢

优化方案：当前中间层协议里已经有草稿的分层信息和轨道的分轨道信息，可以按照层次和轨道分批拉取和渲染资源

注意点：对于用户来说，草稿和轨道的显示是渐进式的，对用户体验有些改变，虽然每次拉取和上屏的数据都比较小，但总的拉取次数和上屏次数，跟最终加载完成的时间是成反比的，这里需要对参数进行不停调优，在不同网络 and 不同配置的机器下，找一个平衡点

草稿云解析

草稿云编辑

高性能架构（未来规划）

单向依赖/高性能分层架构

单向依赖架构除了帮我们构建一个稳定易迭代的系统之外，利用单向依赖架构，可以做许多性能有关的工作。

我们可以利用单向依赖架构，结合高性能分层，把模块至少分为**首屏**，**部分可交互**，**完整可交互**，**插件层**这几层。

然后让应用在不同的生命周期去加载不同层次的代码，高层模块因为不依赖低层模块，我们可以让高层模块优先独立加载和执行。比如视频编辑器未来可以分为数据层、渲染层、feature层、插件层（分享、登录等组件），为了保证用户第一时间看到编辑器和编辑器里的数据，我们先加载和执行最高层的数据层，接下来再去处理渲染层。

这两层加载和执行完之后就可以让用户第一时间看到编辑器里的数据了。

最后在加载页面中的各种插件。在单向依赖架构下，我们可以按照这种方式，很方便的调节各个层级代码的加载和执行，让用户尽早看到页面，将那些不需要第一时间加载的部分很方便的进行延迟加载和执行。

UI&逻辑分离架构

在一些小项目中，UI和逻辑经常是被写在一起的。比如一些我们常见的vue页面。在这些比较小的月抛性项目里算不上特别大的问题。但当项目变大后，UI和逻辑不分离带来的问题也随之放大。

主要问题有这么几个：

- UI和逻辑无法单独被复用

一个需求的逻辑层通常会对应N个UI层，比如一份查找替换的逻辑层，要对应pc、mobile、pad等多个UI层。如果逻辑层耦合了某个端的UI，逻辑层就无法直接复用在其他端。同理，如果UI层耦合了逻辑层，那这些UI组件也将无法再复用在其他项目中。

- UI层和逻辑层的需求迭代频率不一致

通常来说，当核心业务逻辑确定之后，UI的迭代频率比逻辑层要快一些，当然在某些阶段可能也会有逻辑层迭代频率比UI层。

总而言之，UI和逻辑的迭代频率是很难保持一致的。我们在修改逻辑层或者修改UI层的时候，都不希望会影响另外一部分代码。但如果它们是耦合在一起的，不管修改逻辑还是UI，都需要修改同一个模块，以至于有可能影响到对方的代码。

- UI和逻辑的运行环境不一样

UI经常会含有一些宿主环境相关的代码，比如浏览器下的window对象等等，如果UI被耦合在逻辑里，那么这些逻辑代码就不方便运行在一些没有浏览器window对象的环境。比如这时候我们无法顺利在node、worker，或者单元测试中使用这些逻辑代码。

- 逻辑层和渲染层的自动化测试都不方便

逻辑层的单元测试里要去mock各种环境相关的对象，UI层的e2e等测试里也要避免被逻辑层的代码影响。

基于以上原因，UI和逻辑分离是非常必要的。逻辑层一般都是指核心业务逻辑，如草稿计算、函数计算等等，而UI只是这些业务逻辑的一种展示形式。在许多没有UI层的场景下，比如node-ssr、开放平台api，自动化测试这些场景下，业务的逻辑层也需要能独立运行下去。

当逻辑和UI分开之后，我们再考虑如何将逻辑和UI联系起来，比如根据事件、中介者、扩展点、依赖注入等等。

基于UI和逻辑分离架构，我们也可以提高页面打开的整体速度，比如某些组件，先仅仅下载和执行UI相关代码，让用户及早看到页面相关元素，再加载逻辑代码，将逻辑代码注入给UI，完成整个功能。

这种方式适用于UI和逻辑是并列关系的情况，UI和逻辑都不依赖对方的存在，它们可以在后期通过中介者绑定起来。典型的如工具栏模块，我们可以将工具栏分为3个部分：

- UI部分，用workbench配置文件的方式就可以独立的将UI绘制出来
- 异步加载的逻辑代码，比如点击某个按钮后将要执行的feature
- 中间胶水代码，作为中介者将按钮（UI代码）和点击按钮后的事件以及事件里需要执行的逻辑绑定起来（逻辑代码）

在一些只读权限的文档中，因为用户不会触发UI对应的真正点击事件，这种情况下，我们其实只需要完成第一步 - 将工具栏的UI部分渲染出来就可以了。

只读&可编辑分离架构

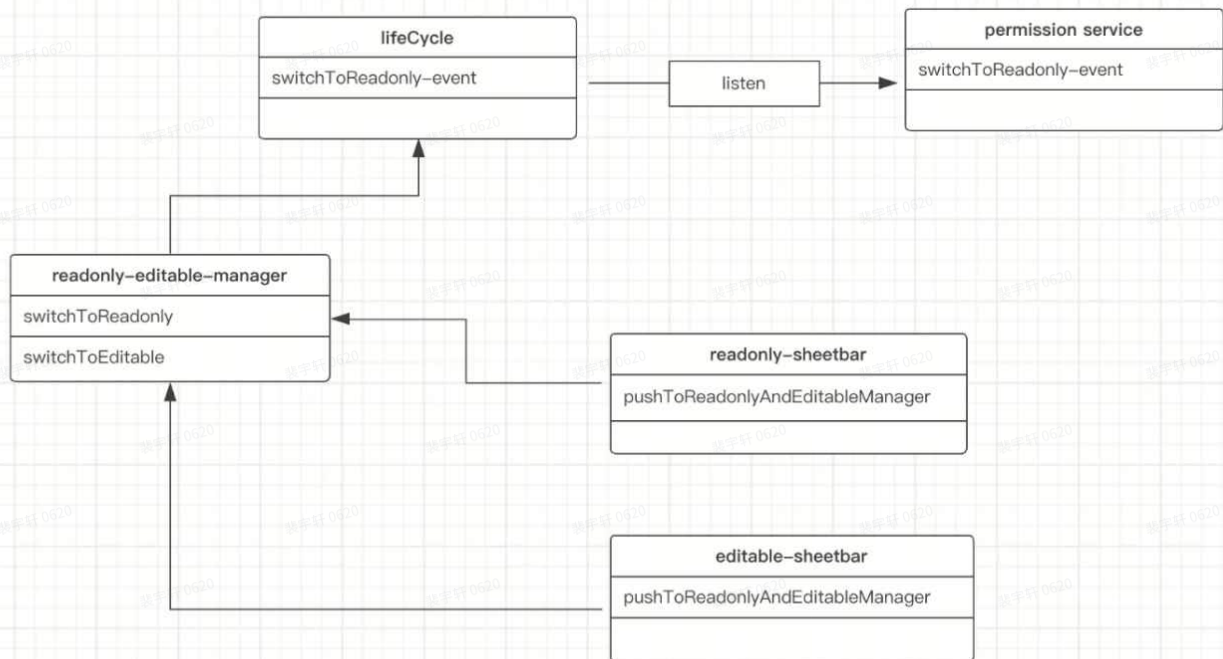
大型编辑器业务在满足业务复杂性同时，保持高性能是不太容易的。页面中存在几百个大的组件，这些组件的全部代码下载下来就占了几十万行

从状态上来看，大部份组件都分为只读和可编辑两种状态，比如未登录状态下可能某些组件是只读的，只有登录态或者在另外一些场景下，这些组件才可以点击或者编辑

这两种状态下需要执行的逻辑是非常不一样的。只读态下需要之行的逻辑和需要下载的代码都要比可编辑态下少的多。如果我们可以把组件的只读和可编辑状态分开，那么某些组件可能在80%的场景只需要加载和执行原本20%的代码。

可以用状态模式 + 装饰者模式来解决只读-可编辑分离的问题。大致思路如下：

- 只读组件优先单独开发，只考虑只读的场景，可编辑的代码、事件等都不需要被包含在只读组件中
- 可编辑代码通过高阶组件给只读组件动态装饰上可编辑功能
- 系统初始化时优先加载只读组件代码，当系统中触发只读-可编辑切换时，给对象组件加载并装饰可编辑代码，并切换到可编辑组件

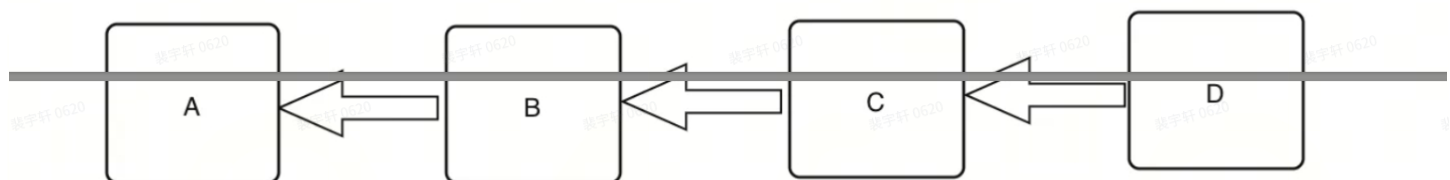


插件化架构

如果整个系统都是基于单向依赖原则去构建。一旦各个模块之间的联系都遵循了单向依赖架构，那整个系统也很容易被改造成插件化架构

从低层往高层看，任何低层模块对于它的高层模块而言，都可以看成高层模块的插件，它们是可选可组装的，低层模块可以像洋葱一样，一层一层被剥离掉，而不影响高层模块的功能，高层模块不需要知道低层模块的存在。

比如当我们剥离掉模块D，这时候A，B，C模块可以脱离D模块在系统中独立运行。同理在一些场景下，我们也可以剥离掉模块C和模块D，让A，B这两个模块独立运行。



多端分离架构

当各个终端的代码混在一起的时候，除了业务逻辑的耦合交叉降低代码可维护性之外，对性能也有非常大的损伤。

```
1
2 if (pc){
3     // showPcTips();
4 }else if (mobile){
5     // showMobileTips();
6 }else if (native){
7     // showNativeTips();
8 }
```

像这些代码里，在pc端要加载mobile端的代码，在mobile端要加载pc端的代码，都会严重增大代码包的体积，我们用依赖注入+多态service的方式来分离不同端的代码，让各个端的代码尽量不要交叉被包含在同一个文件中，对有效提升业务性能也有不错的帮助。

性能防腐系统（未来规划）

业务性能符合熵增定律，随着功能堆砌代码量持续增加，如果不加以管理和治理，性能一定是持续恶化的。主要可以从这几个方面降低性能恶化的速度和可能性

- 优秀的架构，如合理分层、合理的层和模块依赖关系，尽量抹平同步和异步加载等等
- 对性能有良好的开发意识，在开发中能及早关注并解决性能问题
- 完善的性能监控体系，能及时发现并清理导致性能恶化的问题

未来会使用或者开发这些性能监控系统来尽量防止性能随着需求迭代而腐坏

开发分支监控

- 在开发过程中就及时监控到性能问题，如对比开发过程的代码增量，并设置合理的红线阈值
- 对每个新增或者修改的函数跑性能自动化测试，并设置合理的红线阈值

CI流水线监控

- 在CI阶段跑性能自动化, 比如在关键数据上持续对当前分支、主干、现网进行性能对比, 并给出分析和预警报告

性能&稳定性看板监控

当代码发布到外网之后, 需要密切关注现网性能数据变化, 需要建立完善的看板并至少监控以下数据

- 首屏打开性能
- 部分可交互性能
- 完整可交互性能
- 卡顿率
- crash率
- 白屏率
- 内存占用
- cpu占用
- 导入导出速度
- 等等

性能SRE

即网站可靠性观测, 一般会针对现网重点性能&稳定性数据, 设置合理的红线阈值并进行实时告警, 至少包括:

- 首屏打开性能
- 部分可交互性能
- 完整可交互性能
- 卡顿率
- crash率
- 白屏率

竞品数据自动化对比

通过自动化工具, 定期和竞品主要性能数据进行分析对比, 代替周期漫长的人工测试

业界方案大致是在docker中运行 puppeteer 和 lighthouse

一些工具：

Lighthouse

chrome自带的性能分析和打分工具，上手快

Chrome performance

强大的本地性能分析工具，从资源到代码执行都

Wepagetest

可能是目前最好的远程性能分析网站，用无头浏览器模拟用户对目标站点发出请求，可以自由选择网络地区、机型、网速等

自埋点

tea等平台，需要时刻关注埋点数据

📖 新架构性能优化一期

📖 新架构性能优化二期（资源&数据预加载优化）

📖 其他性能优化